

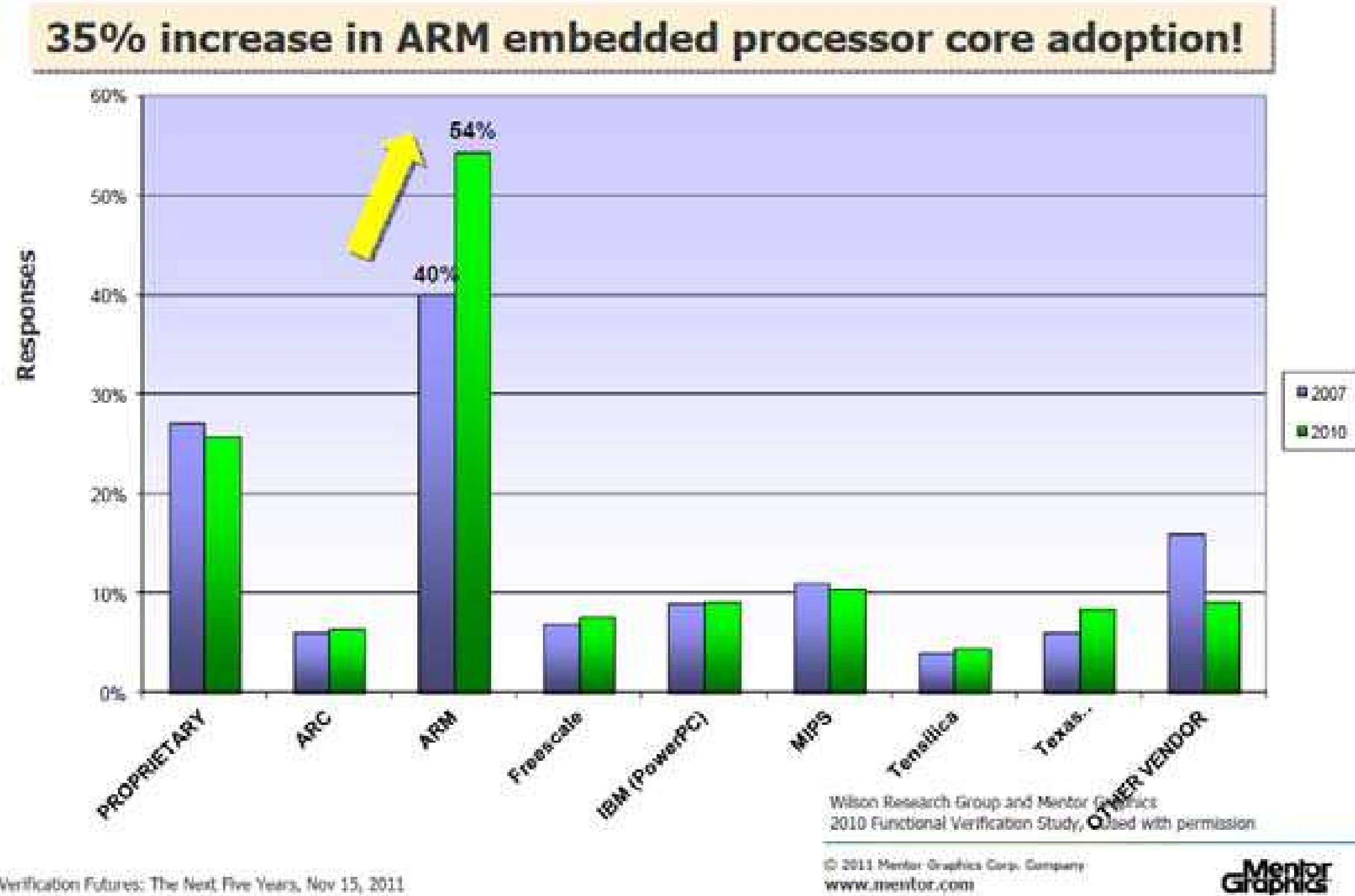
Module: Systèmes Embarqués

Chapitre 3:
Les Processeurs ARM

El Kefi HLEL

Février 2014

MARCHÉ DE L'EMBARQUÉ



Présentation d'ARM

Présentation des Processeurs ARM

- Les **architectures ARM**, développées par *ARM Ltd (*Advanced Risc Machine Limited), sont des architectures RISC 32 bits (ARMv3 à ARMv7) et 64 bits (ARMv8)
- Une particularité des processeurs ARM est leur mode de vente. En effet, ARM Ltd ne fabrique ni ne vend ses processeurs sous forme de circuits intégrés
- La société vend les licences de ses processeurs de manière à ce qu'ils soient gravés dans le silicium par d'autres fabricants. Aujourd'hui, la plupart des grands fondeurs de puces proposent de l'architecture ARM
- De nombreux systèmes d'exploitation sont compatibles avec cette architecture :
  - Symbian S60 avec les Nokia N97 ou Samsung Player HD
  - iOS avec l'iPhone et l'iPad
  - Linux, avec la plupart des distributions ou avec Android
  - BlackBerry OS avec les BlackBerry
  - Windows CE, Windows Phone 7 et Windows RT, une version de windows 8

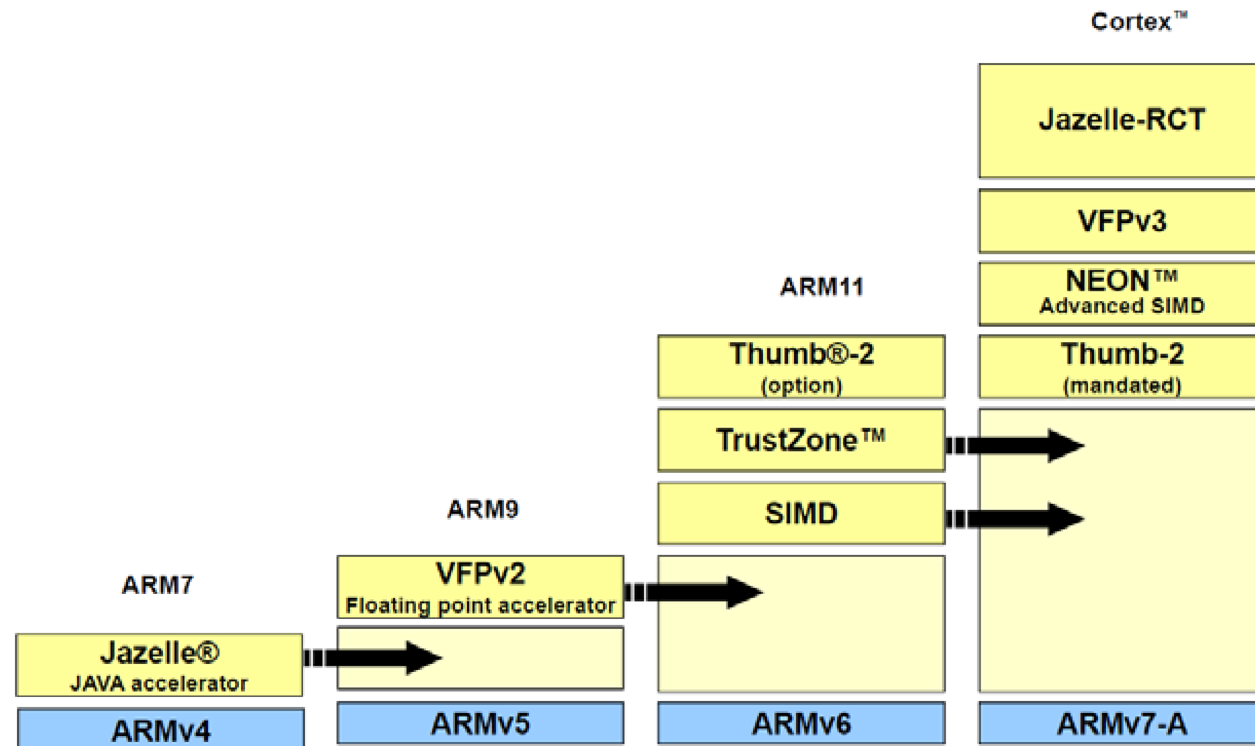
Architectures des Processeurs ARM

Architecture	Famille(s)
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	StrongARM, ARM7TDMI, ARM8, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10E
ARMv6	ARM11
ARMv6-M	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1
ARMv7-A	ARM Cortex-A (ARM Cortex-A8, ARM Cortex-A9 MPCore, ARM Cortex-A5 MPCore, ARM Cortex-A7 MPCore, ARM Cortex-A12 MPCore, ARM Cortex-A15 MPCore)
ARMv7-M	ARM Cortex-M (ARM Cortex-M3)
ARMv7-R	ARM Cortex-R (ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7)
ARMv8	ARM Cortex-A50 (ARM Cortex-A53, ARM Cortex-A57)

Les Processeurs Les Plus Populaire

ARM s'appuie sur de nombreuses architectures différentes afin de toucher un maximum de marchés. Ses processeurs les plus livrés en 2012-2013 sont:

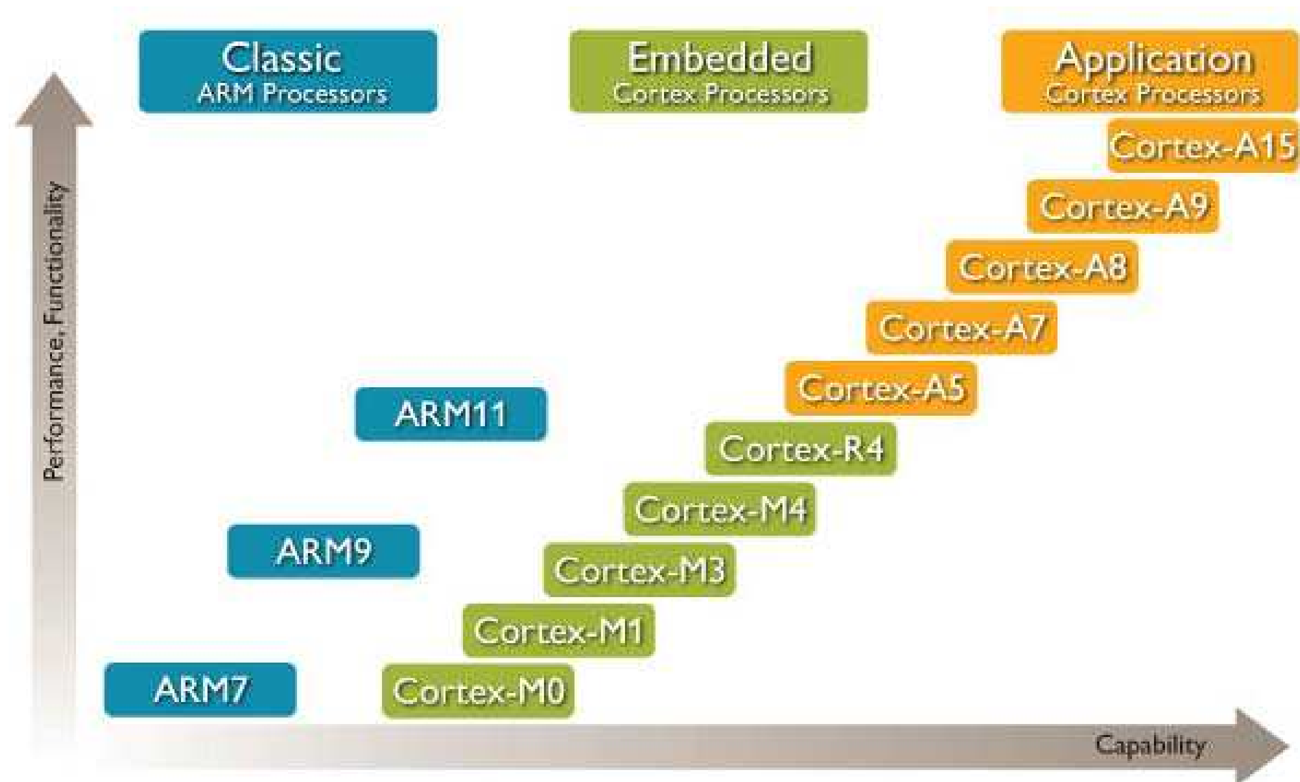
- ARM7: 33% (de livraison)
- ARM9: 19%
- ARM11: 8%
- Cortex-M: 26%
- Cortex-A: 11%
- Cortex-R: 3%



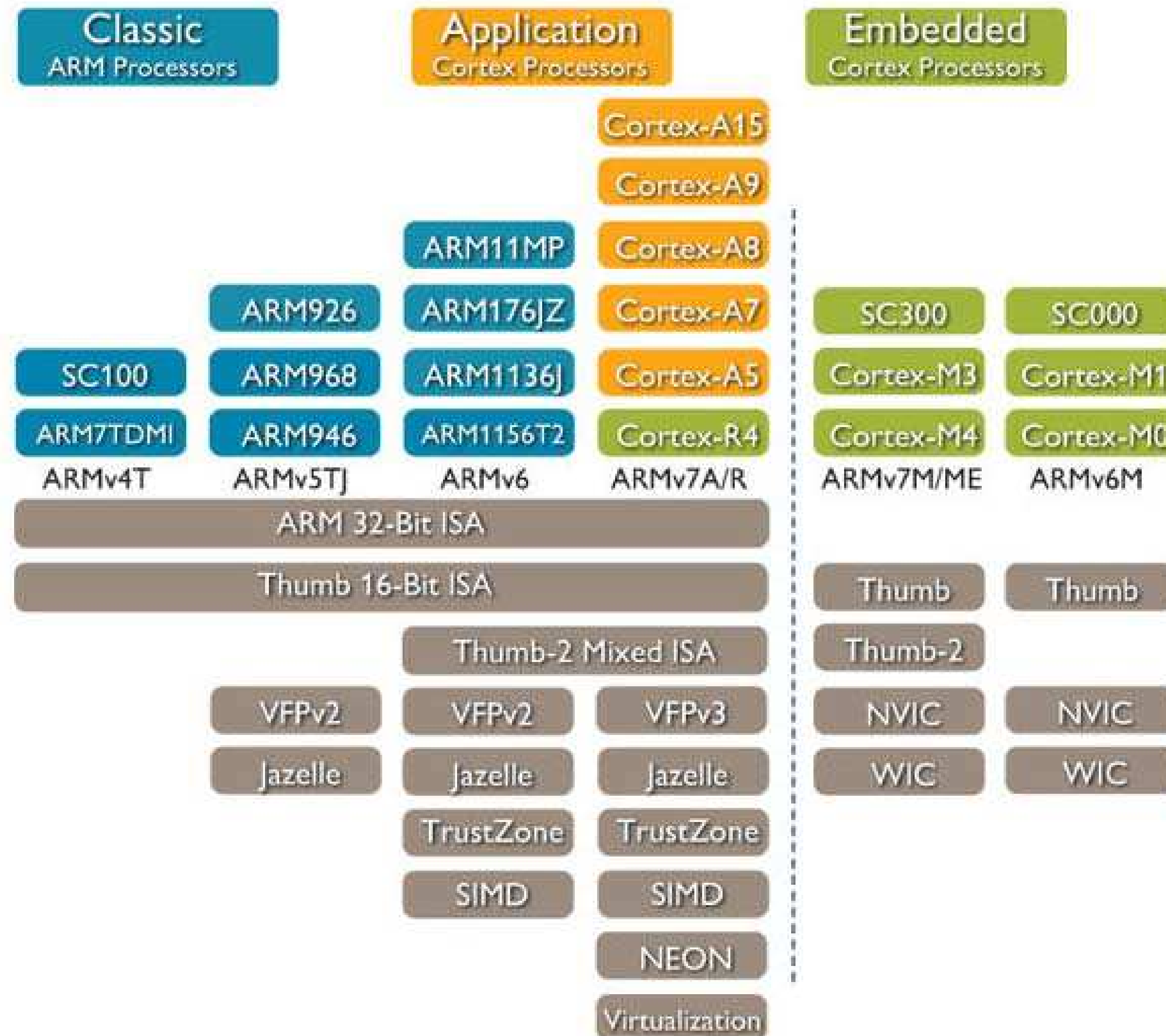
Les Familles Les Plus Populaire

- **Famille ARM7 :**
  - *ARM720T* (MMU)
  - *ARM7TDMI*
  - *ARM7TDMI-S*
  - *ARM7EJ-S* : DSP et Jazelle
- **Famille ARM9** (5 niveaux de pipeline sur les entiers, MMU) : *ARM920T* (double cache de 16 Ko) et *ARM922T* (double cache de 8 Ko)
- **Famille ARM9E**
  - *ARM946E-S* : DSP, double cache, MPU, 1 port AHB
  - *ARM926EJ-S*: DSP, double cache, MMU, 2 ports AHB
  - *ARM966E-S* : DSP, double cache, MPU, 1 ports AHB
- **Famille ARM11** : SIMD, Jazelle, DSP, Thumb-2
  - *ARM1136JF-S* : FPU
  - *ARM1156T2-S*
  - *ARM1156T2F-S* : FPU
  - *ARM1176JZ-S*
  - *ARM1176JZF-S* : FPU
- **Famille Cortex-A**, processeur applicatif: Architecture ARMv7-A, SIMD, Jazelle, DSP, Thumb-2
- **Famille Cortex-R**, processeur temps-réel: Architecture ARMv7-R
- **Famille Cortex-M**, processeur embarqué : Architecture ARMv6-M et ARMv7-M

Caractéristiques (1)



Caractéristiques (2)



Architecture & Jeu d'Instruction (ISA)

ARCHITECTURE ET JEU D'INSTRUCTION

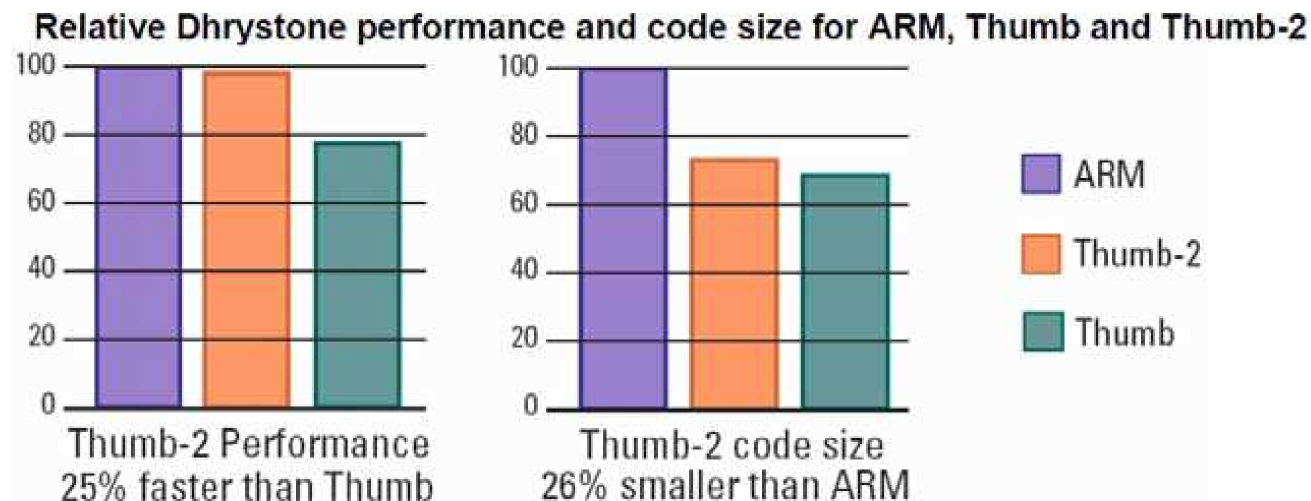
- L'architecture ARM est fortement inspirée des principes de conception RISC. Elle dispose de 16 registres généraux de 32 bits (de r0 à r15). Les instructions codées sur 32 bits jusqu'à l'ARMv7 et sur 64 bits avec l'ARMv8
- Le jeu d'instruction a reçu des extensions au fil du temps, telles que
  - ARM: c'est le jeu d'instruction par défaut sur 32 bits. Il y a 58 instructions dans ce mode et 15 registres + PC
  - Thumb : mode instructions sur 16 bits. Permettant d'améliorer la densité du code. Il y à 30 instructions et 8 registres disponibles + PC. L'activation de ce mode et inversement se fait par l'exécution d'une instruction (respectivement BX et BLX)
  - Jazelle : améliorant l'exécution de byte code Java. Il y a un bit (J) dans le CPSR (Current Program Status Register) qui indique le mode Jazelle. Les instructions sont sur 8 bits
  - NEON, supportant des instructions SIMD (Single Instruction Multiple Data)

THUMB

- **Objectif de thumb**: Jeu d'instructions pour la compression de code
- Moins de registres disponibles (R0-R8) + PC (Program Counter)
- Thumb est un jeu d'instruction 16 bits pour remplacer un sous-ensemble des instructions ARM 32 bits classique et permet un gain important de mémoire
- Les instructions 16 bits améliorent la densité globale d'un programme, bien que certaines opérations nécessitent plus d'instructions. Cela peut réduire le temps de chargement du code en mémoire et augmenter la probabilité de rester dans le cache d'instruction
- Le processeur peut commuter entre les 2 modes (16 et 32 bits). Pour garantir de la performance, on utilise le jeu d'instruction standard sur 32 bits, et le reste Thumb
- **Exemple**: dans le cas d'un système embarqué avec une petite quantité de RAM accessible via un bus de donnée 32 bits mais la majorité est accédé via un second bus de 16 bits. Dans cette situation, il est très intéressant de compiler le programme en mode Thumb et d'optimiser les sections les plus consommatrices en utilisant le jeu d'instructions complet ARM 32 bits, permettant de placer ces instructions plus larges dans le bus d'accès mémoire 32 bits

THUMB-2

- Introduction de nouvelles instructions (130 instructions supplémentaires) pour les manipulations de bits
  - Opération d'entrées/sorties
  - Modifications de bits pour des structures temps réel
- Éviter de commuter entre le mode ARM et Thumb, une application peut être écrite complètement en Thumb2 (exemple avec Cortex M3)
- Thumb-2 (un jeu d'instructions de largeur variable) comprend:
  - Le jeu d'instructions ARM 32 bits
  - Le jeu d'instructions limité 16 bits de Thumb
  - Un ensemble des instructions 32 bits additionnelles
- Thumb-2 a pour but d'atteindre une densité de code proche de Thumb tout en conservant des performances similaires au jeu d'instructions ARM en mémoire 32 bits



JAZELLE

- Jazelle est une technique permettant d'exécuter directement du Bytecode Java dans les architectures ARM comme un troisième état d'exécution (et jeu d'exécution), en parallèle à l'ARM existant et au mode Thumb (ou Thumb-2)
- Le support pour cet état est signalé par le « *J* » dans les architectures (ARMv5TEJ) ou dans les noms de cœurs (exemple: ARM9EJ-S et ARM7EJ-S)
- Le **bytecode:** est un code intermédiaire entre les instructions machines et le code source, il n'est pas directement exécutable. Il est produit par un compilateur
- Le **bytecode Java:** est un bytecode destiné à regrouper des instructions exécutables par une machine virtuelle java. Il désigne un flux d'octets binaire au format d'une classe java. Ce flux est habituellement le résultat de la compilation d'un code source. Ce bytecode peut être exécuté sous de nombreux systèmes d'exploitation par une machine virtuelle Java

NEON: ADVANCED SIMD

- Advanced SIMD (Single Instruction Multiple Data) également appelée NEON ou ("MPE" pour Media Processing Engine – moteur de calcul de médias) combine des jeux d'instructions 64 et 128 bits, qui fournissent de l'accélération de calcul standardisé pour les application de médias, 2D/3D et de traitement du signal
- NEON est inclus dans tous les cœurs Cortex-A8 mais est optionnel dans les Cortex-A9
- Il comporte 32 registres 64 bits flottants (partagé avec le FPU) pouvant être couplés pour former 16 registres de 128 bits flottants, et accepte aussi que ces registres soient traités comme des entiers signés ou non signés de 8, 16, 32 et 64 bits

Fabricants de Processeurs ARM

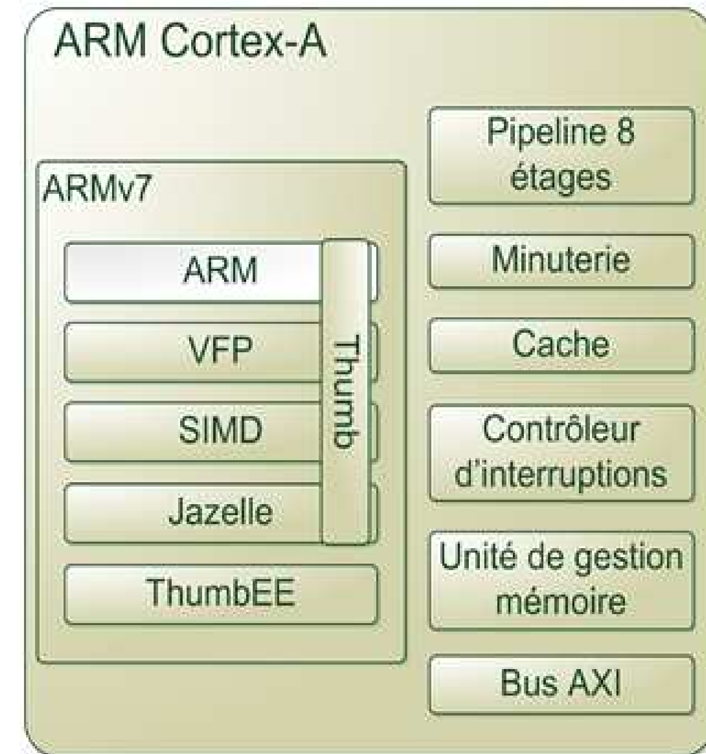
La propriété intellectuelle appartient à une société britannique (ARM Ltd), mais les processeurs sont fabriqués sous licence par différentes entreprises. Parmi les entreprises fabriquant les modèles des séries Cortex (les plus avancées), la majorité se trouve en Asie, suivie par les États-Unis et enfin par l'Europe:

- Atmel, États-Unis
- Broadcom, États-Unis (Cortex A9)
- Freescale, États-Unis
- Fujitsu, Japon (cortex M3, Cortex A9, A7, A15)
- HiSilicon, Chine (Cortex A9, A7, A15)
- Huawei Technologies, Chine (Cortex A9, A7, A15)
- Infineon, Allemagne
- LG Electronics, Corée du Sud (A50)
- NXP, Pays-Bas
- Nvidia, États-Unis (Cortex A9)
- Qualcomm, États-Unis
- Samsung, Corée du Sud (Cortex A8, Cortex A9, A15 (Exynos 5250), A7)
- STMicroelectronics, France & Italie (Cortex A9, Cortex A15)
- Texas Instruments, États-Unis (Cortex A9, A15)
- Toshiba, Japon
- Xilinx, États-Unis (FPGA Cortex-A8)

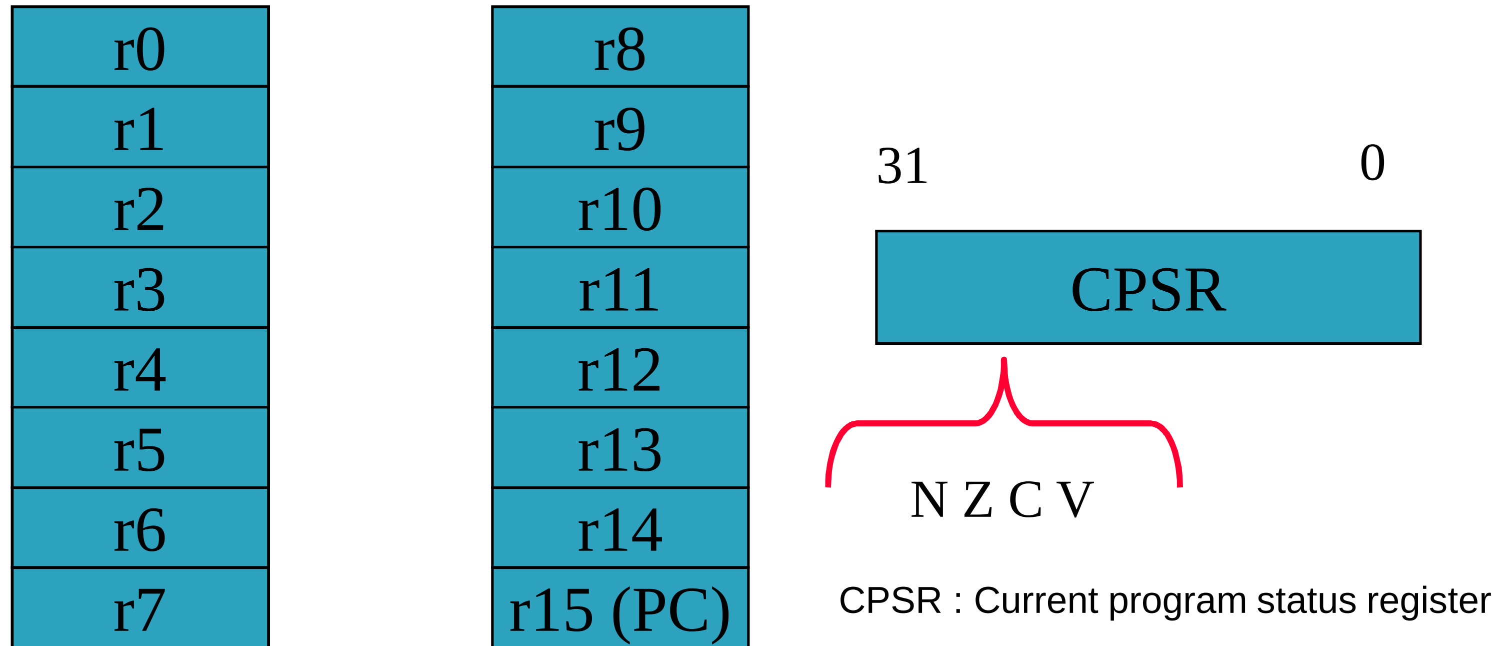
EXEMPLES

EXEMPLE: ARMV7

- ARMv7
  - ARM: 234 instructions
  - SIMD/VFP: 143 instructions
  - Thumb-2: 218 instructions
- La famille Cortex: Famille de processeurs partageants le même Core 32 bits et le même jeu d'instruction de base ARMv7
- Cortex-A: haute performance, faible consommation
  - Applications : jeux, communication sans fil, routers, multimédia embarqué
- Cortex-R: applications temps réels
  - Applications : automobile exigeant des temps de réponses fixe
- Cortex-M: applications microcontrôleurs, plus de fonctionnalités ou variés



ETUDE DE L'ARCHITECTURE ARM7



- N (Negative), Z (Zero), C (Carry), V (oVerflow)
- mode – control processor mode
- T – control instruction set
  - T = 1 – instruction stream is 16-bit Thumb instructions
  - T = 0 – instruction stream is 32-bit ARM instructions
- I F – interrupt enables



Instructions (en assembleur)

- Basic format:

  ADD r0,r1,r2

  - Computes r1+r2, stores in r0
- Immediate operand:

  ADD r0,r1,#2

  - Computes r1+2, stores in r0
- ADD, ADC : add (w. carry)
- SUB, SBC : subtract (w. carry)
- RSB, RSC : reverse subtract (w. carry)
- MUL, MLA : multiply (and accumulate)
- AND, ORR, EOR
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ASL, ASR : arithmetic shift left/right
- ROR : rotate right
- RRX : rotate right extended with C
- MOV, MVN : move (negated)

  MOV r0, r1 ; r0 = r1

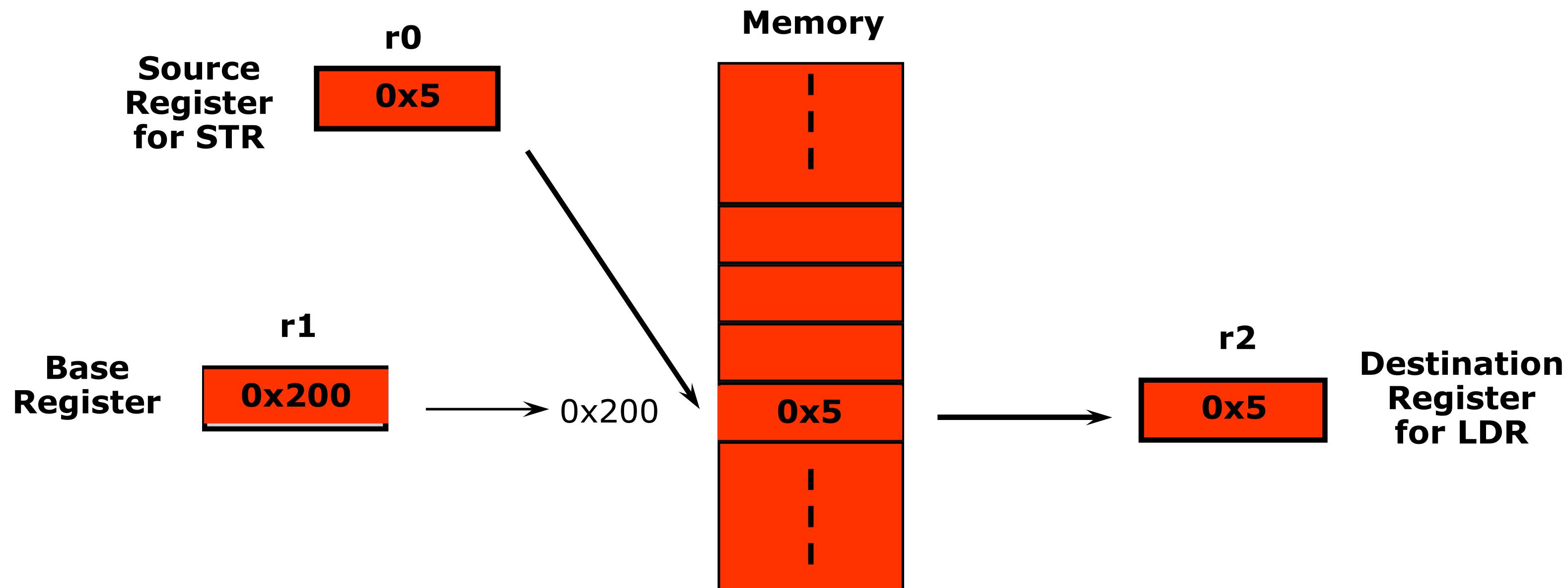
  MVN r0, r1 ; r0 = -r1

ARM load/store instructions

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:
  - register indirect : LDR r0,[r1]
    - R0 = MC[r1]
  - with second register : LDR r0,[r1,-r2]
  - with constant : LDR r0,[r1,#4]

Exemple: Load and Store Word or Byte

- The memory location to be accessed is held in a base register
  - STR r0, [r1] → Store contents of r0 to location pointed to by contents of r1
  - LDR r2, [r1] → Load r2 with contents of memory location pointed to by contents of r1



Exemples 1 & 2: C et ARM

- **En C:**

  x = (a + b) - c;

- **En Assembler ARM:**

```
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a
ADR r4,b          ; get address for b, reusing r4
LDR r1,[r4]       ; get value of b
ADD r3,r0,r1      ; compute a+b
ADR r4,c          ; get address for c
LDR r2[r4]        ; get value of c
SUB r3,r3,r2      ; complete computation of x
ADR r4,x          ; get address for x
STR r3, [r4]      ; store value of x
```

- **En C:**

```
If (a==b)
    { a++; }
    else
     { a--; }
```

- **En Assembleur ARM:**

```
CMP  R1,R2        /* a dans R1, b dansR2*/
ADDEQ R1,#1      /* si égale ajouter 1*/
SUBNE R1,#1      /* soustraire 1 si non */
```

EXEMPLE 3: IF

- **En C:**

  if (a > b) { x = 5; y = c + d; } else x = c - d;

- **En Assembleur:**

```
; compute and test condition
    ADR r4,a            ; get address for a
    LDR r0,[r4]         ; get value of a
    ADR r4,b            ; get address for b
    LDR r1,[r4]         ; get value for b
    CMP r0,r1           ; compare a < b
    BGE fblock          ; if a >= b, branch to false block
; true block
    MOV r0,#5           ; generate value for x
    ADR r4,x            ; get address for x
    STR r0,[r4]         ; store x
    ADR r4,c            ; get address for c
    LDR r0,[r4]         ; get value of c
    ADR r4,d            ; get address for d
    LDR r1,[r4]         ; get value of d
    ADD r0,r0,r1        ; compute y
    ADR r4,y            ; get address for y
    STR r0,[r4]         ; store y
    B after             ; branch around false block
```

```
; false block
fblock    ADR r4,c        ; get address for c
          LDR r0,[r4]     ; get value of c
          ADR r4,d        ; get address for d
          LDR r1,[r4]     ; get value for d
          SUB r0,r0,r1    ; compute c-d
          ADR r4,x        ; get address for x
          STR r0,[r4]     ; store value of x
after ...
```

EXEMPLE 4 : FILTRE FIR

- **En C**

```
for (i=0, f=0; i<N; i++)
    f = f + c[i]*x[i];
```

- **En Assembleur**

```
; loop initiation code
    MOV r0,#0          ; use r0 for i
    MOV r8,#0          ; use separate index for arrays
    ADR r2,N           ; get address for N
    LDR r1,[r2]        ; get value of N
    MOV r2,#0          ; use r2 for f
    ADR r3,c           ; load r3 with base of c
    ADR r5,x           ; load r5 with base of x
; loop body
loop LDR r4,[r3,r8]    ; get c[i]
    LDR r6,[r5,r8]     ; get x[i]
    MUL r4,r4,r6       ; compute c[i]*x[i]
    ADD r2,r2,r4       ; add into running sum
    ADD r8,r8,#4       ; add one word offset to array index
    ADD r0,r0,#1       ; add 1 to i
    CMP r0,r1          ; exit?
    BLT loop           ; if i < N, continue
```